Java, **Data Structures** are ways to store, organize, and manage data efficiently so it can be used effectively in programs.

**1. Categories of Data Structures in Java**

**A. Linear Data Structures**

(Data arranged in sequence — elements have next & previous)

* **Array** → Fixed-size, index-based storage.
* **ArrayList** → Dynamic array (resizable).
* **LinkedList** → Nodes connected by references, easy insertion/deletion.
* **Stack** → LIFO (Last In, First Out) order.
* **Queue** → FIFO (First In, First Out) order.
* **Deque** → Double-ended queue.

**B. Non-Linear Data Structures**

(Data arranged in a hierarchy or interconnected)

* **Tree** → Hierarchical structure (e.g., BinaryTree, BinarySearchTree).
* **Graph** → Nodes connected with edges, used for networks, maps, etc.

**C. Hash-Based Data Structures**

(Uses hashing for fast access)

* **HashMap** → Key-value pairs, fast lookup.
* **HashSet** → Stores unique elements, unordered.
* **LinkedHashMap** → Key-value pairs in insertion order.
* **LinkedHashSet** → Unique elements in insertion order.

**2. Java Collections Framework (JCF)**

The JCF is divided into **interfaces** and **classes**:

| **Interface** | **Implementations** | **Purpose** |
| --- | --- | --- |
| **List** | ArrayList, LinkedList, Vector, Stack | Ordered, allows duplicates |
| **Set** | HashSet, LinkedHashSet, TreeSet | Unordered/Ordered, no duplicates |
| **Queue** | LinkedList, PriorityQueue, ArrayDeque | FIFO / priority order |
| **Map** | HashMap, LinkedHashMap, TreeMap | Key-value pairs |

**3. Example Program – Using Different Data Structures**

import java.util.\*;

public class DataStructureExample {

public static void main(String[] args) {

// ArrayList Example

List<String> names = new ArrayList<>();

names.add("Akila");

names.add("Karthikeyan");

names.add("Priya");

System.out.println("ArrayList: " + names);

// LinkedList Example

LinkedList<Integer> numbers = new LinkedList<>();

numbers.add(10);

numbers.add(20);

numbers.add(30);

System.out.println("LinkedList: " + numbers);

// Stack Example

Stack<String> stack = new Stack<>();

stack.push("Java");

stack.push("Python");

stack.push("C++");

System.out.println("Stack (pop): " + stack.pop());

// Queue Example

Queue<String> queue = new LinkedList<>();

queue.add("Task1");

queue.add("Task2");

System.out.println("Queue (poll): " + queue.poll());

// HashMap Example

Map<Integer, String> students = new HashMap<>();

students.put(101, "Akila");

students.put(102, "Karthikeyan");

System.out.println("HashMap: " + students);

// HashSet Example

Set<String> cities = new HashSet<>();

cities.add("Chennai");

cities.add("Madurai");

cities.add("Chennai"); // Duplicate ignored

System.out.println("HashSet: " + cities);

}

}

**Array**

* **What:** Fixed-size, index-based storage for elements of the same type.
* **Why:** Very fast access (O(1) by index), but size cannot be changed after creation.
* **Example:**

public class ArrayExample {

public static void main(String[] args) {

int[] arr = {10, 20, 30, 40};

for (int i = 0; i < arr.length; i++) {

System.out.println("Element at index " + i + ": " + arr[i]);

}

}

}

**2. ArrayList**

* **What:** A resizable array (part of java.util), automatically grows/shrinks.
* **Why:** More flexible than arrays, easy to add/remove elements.
* **Example:**

import java.util.ArrayList;

public class ArrayListExample {

public static void main(String[] args) {

ArrayList<String> names = new ArrayList<>();

names.add("Akila");

names.add("Karthikeyan");

names.add("Priya");

System.out.println("Names: " + names);

names.remove("Priya");

System.out.println("After removal: " + names);

}

}

**3. LinkedList**

* **What:** Elements (nodes) connected via references (pointers) to the next/previous node.
* **Why:** Faster insertion/deletion (O(1)) compared to ArrayList (O(n)), but slower access.
* **Example:**

import java.util.LinkedList;

public class LinkedListExample {

public static void main(String[] args) {

LinkedList<Integer> list = new LinkedList<>();

list.add(10);

list.add(20);

list.add(30);

list.addFirst(5); // Add at beginning

list.addLast(40); // Add at end

System.out.println("LinkedList: " + list);

}

}

**4. Stack**

* **What:** LIFO (Last In, First Out) — the last element added is the first removed.
* **Why:** Useful for undo features, expression evaluation, backtracking.
* **Example:**

import java.util.Stack;

public class StackExample {

public static void main(String[] args) {

Stack<String> stack = new Stack<>();

stack.push("A");

stack.push("B");

stack.push("C");

System.out.println("Stack: " + stack);

System.out.println("Popped: " + stack.pop());

System.out.println("Top Element: " + stack.peek());

}

}

**5. Queue**

* **What:** FIFO (First In, First Out) — the first element added is the first removed.
* **Why:** Useful for task scheduling, order processing.
* **Example:**

import java.util.LinkedList;

import java.util.Queue;

public class QueueExample {

public static void main(String[] args) {

Queue<Integer> queue = new LinkedList<>();

queue.add(1);

queue.add(2);

queue.add(3);

System.out.println("Queue: " + queue);

System.out.println("Removed: " + queue.poll());

System.out.println("Queue after removal: " + queue);

}

}

**6. Deque (Double-Ended Queue)**

* **What:** Elements can be added/removed from both ends.
* **Why:** More flexible than queue/stack — supports both LIFO and FIFO.
* **Example:**

import java.util.Deque;

import java.util.LinkedList;

public class DequeExample {

public static void main(String[] args) {

Deque<String> deque = new LinkedList<>();

deque.addFirst("First");

deque.addLast("Last");

deque.addFirst("New First");

System.out.println("Deque: " + deque);

deque.removeLast();

System.out.println("After removing last: " + deque);

}

}

L **Linear Data Structures** with **real-time examples**

**1. Array**

**Definition:**

* Fixed-size collection of elements stored in contiguous memory locations.
* Access using **index**.

💡 **Real-time Example:**

* **Seating arrangement in a cinema hall** → Each seat has a fixed position (like index numbers in an array).
* Example: Seat[0], Seat[1], etc.

String[] seats = {"A1", "A2", "A3", "A4"};

System.out.println(seats[2]); // Output: A3

**2. ArrayList**

**Definition:**

* A **dynamic array** — it can **grow** and **shrink** at runtime.

**Real-time Example:**

* **Shopping cart** in an e-commerce app → Items can be added or removed anytime.

import java.util.ArrayList;

ArrayList<String> cart = new ArrayList<>();

cart.add("Laptop");

cart.add("Mouse");

cart.add("Keyboard");

System.out.println(cart);

**3. LinkedList**

**Definition:**

* Elements are **nodes** connected with links (pointers).
* Good for frequent **insertion & deletion**.

**Real-time Example:**

* **Music playlist** → Songs are linked; you can add/remove in the middle without shifting everything.

import java.util.LinkedList;

LinkedList<String> playlist = new LinkedList<>();

playlist.add("Song A");

playlist.add("Song B");

playlist.add("Song C");

System.out.println(playlist);

**4. Stack (LIFO)**

**Definition:**

* Last In, First Out — the last element added is the first removed.

**Real-time Example:**

* **Undo feature** in a text editor — the most recent action is undone first.

import java.util.Stack;

Stack<String> undoStack = new Stack<>();

undoStack.push("Type A");

undoStack.push("Type B");

undoStack.push("Type C");

System.out.println(undoStack.pop()); // Removes "Type C"

**5. Queue (FIFO)**

**Definition:**

* First In, First Out — first element added is the first removed.

**Real-time Example:**

* **Ticket counter queue** — first person in line gets served first.

import java.util.LinkedList;

import java.util.Queue;

Queue<String> queue = new LinkedList<>();

queue.add("Person 1");

queue.add("Person 2");

System.out.println(queue.poll()); // Removes "Person 1"

**6. Deque (Double-ended Queue)**

**Definition:**

* You can add or remove elements from **both ends**.

**Real-time Example:**

* **Browser history navigation** — move forward or backward.

import java.util.Deque;

import java.util.LinkedList;

Deque<String> deque = new LinkedList<>();

deque.addFirst("Page 1");

deque.addLast("Page 2");

System.out.println(deque);

## ****Java Program: Real-time Examples of Linear Data Structures****

import java.util.\*;

public class LinearDataStructureExamples {

public static void main(String[] args) {

// 1. Array Example – Student Marks

int[] studentMarks = {85, 90, 78, 88, 76};

System.out.println(" Array Example (Student Marks):");

for (int i = 0; i < studentMarks.length; i++) {

System.out.println("Student " + (i+1) + " Marks: " + studentMarks[i]);

}

// 2. ArrayList Example – Grocery List

ArrayList<String> groceryList = new ArrayList<>();

groceryList.add("Milk");

groceryList.add("Bread");

groceryList.add("Eggs");

System.out.println("\n ArrayList Example (Grocery List): " + groceryList);

// 3. LinkedList Example – Train Coaches

LinkedList<String> train = new LinkedList<>();

train.add("Engine");

train.add("Coach A");

train.add("Coach B");

train.add("Coach C");

System.out.println("\n LinkedList Example (Train Coaches): " + train);

train.addFirst("New Engine");

System.out.println("After adding at start: " + train);

// 4. Stack Example – Browser History (LIFO)

Stack<String> browserHistory = new Stack<>();

browserHistory.push("Google");

browserHistory.push("YouTube");

browserHistory.push("ChatGPT");

System.out.println("\n Stack Example (Browser History): " + browserHistory);

browserHistory.pop();

System.out.println("After going back: " + browserHistory);

// 5. Queue Example – Customer Service (FIFO)

Queue<String> customerQueue = new LinkedList<>();

customerQueue.offer("Customer 1");

customerQueue.offer("Customer 2");

customerQueue.offer("Customer 3");

System.out.println("\nQueue Example (Customer Service): " + customerQueue);

customerQueue.poll(); // Serve first customer

System.out.println("After serving one: " + customerQueue);

// 6. Deque Example – Train Boarding (Double-ended Queue)

Deque<String> trainBoarding = new ArrayDeque<>();

trainBoarding.addFirst("Passenger 1");

trainBoarding.addLast("Passenger 2");

trainBoarding.addFirst("Passenger 3");

System.out.println("\n Deque Example (Train Boarding): " + trainBoarding);

trainBoarding.removeLast();

System.out.println("After removing from last: " + trainBoarding);

}

}

## ****1. What is a Non-Linear Data Structure?****

A **non-linear data structure** is one where data elements are **not arranged sequentially**.  
Instead, elements are connected in **hierarchical** (tree-like) or **network** (graph-like) fashion.

🔹 In **linear DS**, you move from one element to the next in order.  
🔹 In **non-linear DS**, you can have multiple paths and connections.

## ****2. Why Use Non-Linear Data Structures?****

We use them when:

* **Data has a hierarchy** (e.g., folder structure in a computer)
* **Relationships between data are complex** (e.g., social networks, maps)
* **Searching or accessing specific data needs to be fast**
* **Modeling real-world relationships** (e.g., family tree, city map)

## ****3. Common Types of Non-Linear Data Structures****

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Tree** | Hierarchical structure with parent-child relationships. | File system |
| **Graph** | Network of nodes connected by edges. | Social media connections |
| **Heap** | Special tree-based structure for priority. | Job scheduling |

## ****4. Real-Life Examples****

1. **Tree (Hierarchical)** → Computer file system (Folders inside folders)
2. **Graph (Network)** → Google Maps (Cities connected by roads)
3. **Heap (Priority Queue)** → Hospital emergency room (patients treated by urgency)

## ****5. Purpose of Non-Linear DS****

* Fast searching & retrieval in hierarchical data.
* Better representation of **complex relationships**.
* Optimized performance for certain operations (like shortest path in graphs).
* Used in **databases**, **AI algorithms**, **networking**, and **game development**.

## ****6. Implementation Examples in Java****

### ****Tree Example****

class Node {

String data;

Node left, right;

Node(String data) {

this.data = data;

left = right = null;

}

}

public class TreeExample {

public static void main(String[] args) {

Node root = new Node("Root");

root.left = new Node("Left Child");

root.right = new Node("Right Child");

System.out.println("Root Node: " + root.data);

System.out.println("Left Child: " + root.left.data);

System.out.println("Right Child: " + root.right.data);

}

}

### ****Graph Example (Adjacency List)****

import java.util.\*;

public class GraphExample {

private Map<String, List<String>> graph = new HashMap<>();

public void addEdge(String from, String to) {

graph.computeIfAbsent(from, k -> new ArrayList<>()).add(to);

}

public void displayGraph() {

for (String node : graph.keySet()) {

System.out.println(node + " -> " + graph.get(node));

}

}

public static void main(String[] args) {

GraphExample g = new GraphExample();

g.addEdge("A", "B");

g.addEdge("A", "C");

g.addEdge("B", "D");

g.displayGraph();

}

}

**Real-life Examples**

**(a) Tree Example – File System**

class TreeExample {

static class Node {

String data;

Node left, right;

Node(String data) {

this.data = data;

left = right = null;

}

}

public static void main(String[] args) {

Node root = new Node("Root Folder");

root.left = new Node("Documents");

root.right = new Node("Pictures");

root.left.left = new Node("Resume.docx");

root.left.right = new Node("Notes.txt");

root.right.left = new Node("Vacation.jpg");

root.right.right = new Node("Family.png");

System.out.println("Root: " + root.data);

System.out.println("Left Child: " + root.left.data);

System.out.println("Right Child: " + root.right.data);

}

}

💡 **Real life:** Your PC’s folder system is a **tree**.

**(b) Graph Example – Social Network**

import java.util.\*;

class GraphExample {

private Map<String, List<String>> graph = new HashMap<>();

public void addFriend(String person, String friend) {

graph.computeIfAbsent(person, k -> new ArrayList<>()).add(friend);

}

public void showNetwork() {

for (String person : graph.keySet()) {

System.out.println(person + " → " + graph.get(person));

}

}

public static void main(String[] args) {

GraphExample g = new GraphExample();

g.addFriend("Alice", "Bob");

g.addFriend("Alice", "Charlie");

g.addFriend("Bob", "David");

g.showNetwork();

}

}

💡 **Real life:** Facebook friend suggestions work on **graph traversal**.

**(c) Heap Example – Job Scheduling**

import java.util.PriorityQueue;

class HeapExample {

static class Job implements Comparable<Job> {

String name;

int priority; // smaller = higher priority

Job(String name, int priority) {

this.name = name;

this.priority = priority;

}

public int compareTo(Job other) {

return Integer.compare(this.priority, other.priority);

}

}

public static void main(String[] args) {

PriorityQueue<Job> pq = new PriorityQueue<>();

pq.add(new Job("Email Processing", 3));

pq.add(new Job("Data Backup", 1));

pq.add(new Job("Virus Scan", 2));

while (!pq.isEmpty()) {

Job job = pq.poll();

System.out.println("Processing: " + job.name + " (Priority: " + job.priority + ")");

}

}

}

💡 **Real life:** CPU scheduling in an OS uses **heaps**.

## ****1. What They Are****

Hash-based data structures in Java store elements using **hashing** — a technique that converts data into a numerical **hash code**, which decides where it’s stored in memory.  
This allows **very fast search, insert, and delete** operations — often **O(1)** on average.

## ****2. Why We Use Them****

* **Speed:** Average O(1) lookup.
* **Uniqueness:** Avoid duplicate data (HashSet).
* **Key-based Access:** Directly find values using keys (HashMap).
* **Order Preservation:** Linked variants preserve insertion order.

## ****Real-Life Examples****

| **Data Structure** | **Real-Life Example** |
| --- | --- |
| **HashMap** | Dictionary (Word → Meaning) |
| **HashSet** | Employee IDs (no duplicates) |
| **LinkedHashMap** | Attendance register (order of arrival matters) |
| **LinkedHashSet** | List of visited websites (no duplicates, in order) |

✅ **Purpose of Hash-Based DS:**

* **Fast access** to data
* **Avoid duplicates** where necessary
* **Key-value mapping** for organized storage
* **Order tracking** if needed (Linked variants)

## ****How to Implement****

## ****1. HashMap****

### ****What it is****

* Stores **key-value pairs**.
* **No duplicate keys** allowed.
* **Order not guaranteed**.
* **Fast lookup** because of hashing.

### ****Why use it****

* When you want to **quickly find a value** using a key.
* Good for **dictionary-like structures**.

### ****Real-life example****

* **Dictionary app**: Word → Meaning.
* **Student database**: Roll number → Student details.

### ****Example program****

import java.util.HashMap;

public class HashMapExample {

public static void main(String[] args) {

HashMap<Integer, String> students = new HashMap<>();

// Adding key-value pairs

students.put(101, "Akila");

students.put(102, "Karthikeyan");

students.put(103, "Priya");

// Accessing value by key

System.out.println("Roll 102: " + students.get(102));

// Iterating through the map

for (Integer roll : students.keySet()) {

System.out.println("Roll: " + roll + ", Name: " + students.get(roll));

}

}

}

### ****Example:**** HashMap

import java.util.HashMap;

public class HashMapExample {

public static void main(String[] args) {

HashMap<String, Integer> stock = new HashMap<>();

// Adding elements

stock.put("Apple", 50);

stock.put("Banana", 30);

stock.put("Orange", 20);

// Accessing elements

System.out.println("Apple stock: " + stock.get("Apple"));

// Checking key existence

if (stock.containsKey("Banana")) {

System.out.println("Banana is available!");

}

// Iterating

for (String fruit : stock.keySet()) {

System.out.println(fruit + " -> " + stock.get(fruit));

}

}}

## ****2. HashSet****

### ****What it is****

* Stores **only unique elements**.
* **No duplicate values** allowed.
* **No guaranteed order**.

### ****Why use it****

* When you want to store a **collection without duplicates**.
* Fast **search and insert** operations.

### ****Real-life example****

* Storing **unique usernames** in a system.
* Maintaining **unique product IDs**.

### ****Example program****

import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

HashSet<String> usernames = new HashSet<>();

usernames.add("Akila");

usernames.add("Karthik");

usernames.add("Akila"); // Duplicate, will not be added

for (String name : usernames) {

System.out.println(name);

}

}

}

### ****Example:**** HashSet

import java.util.HashSet;

public class HashSetExample {

public static void main(String[] args) {

HashSet<String> cities = new HashSet<>();

cities.add("Chennai");

cities.add("Mumbai");

cities.add("Chennai"); // Duplicate ignored

System.out.println(cities); // Order not guaranteed

}

}

## ****3. LinkedHashMap****

### ****What it is****

* Same as **HashMap**, but **maintains insertion order**.
* Key-value pairs stored in a **linked list** internally.

### ****Why use it****

* When you want **fast lookup** + **maintain the order in which entries were added**.

### ****Real-life example****

* **Recent search history** in browsers.
* **Order-sensitive key-value storage**.

### ****Example program****

import java.util.LinkedHashMap;

public class LinkedHashMapExample {

public static void main(String[] args) {

LinkedHashMap<Integer, String> books = new LinkedHashMap<>();

books.put(1, "Java Basics");

books.put(2, "Spring Boot");

books.put(3, "Data Structures");

for (Integer id : books.keySet()) {

System.out.println("Book ID: " + id + ", Title: " + books.get(id));

}

}

}

### ****Example:**** LinkedHashMap

import java.util.LinkedHashMap;

public class LinkedHashMapExample {

public static void main(String[] args) {

LinkedHashMap<Integer, String> students = new LinkedHashMap<>();

students.put(101, "Akila");

students.put(102, "Karthikeyan");

students.put(103, "Priya");

System.out.println(students); // Maintains insertion order

}

}

## ****4. LinkedHashSet****

### ****What it is****

* Same as **HashSet**, but **maintains insertion order**.
* Stores **unique elements**.

### ****Why use it****

* When you want **no duplicates** + **order maintained**.

### ****Real-life example****

* Storing **ordered unique items** like playlist songs without repetition.

### ****Example program****

import java.util.LinkedHashSet;

public class LinkedHashSetExample {

public static void main(String[] args) {

LinkedHashSet<String> playlist = new LinkedHashSet<>();

playlist.add("Song A");

playlist.add("Song B");

playlist.add("Song C");

playlist.add("Song A"); // Duplicate ignored

for (String song : playlist) {

System.out.println(song);

}

}

}

import java.util.LinkedHashSet;

public class RecentlyVisitedCities {

public static void main(String[] args) {

LinkedHashSet<String> visitedCities = new LinkedHashSet<>();

// Visiting cities

visitedCities.add("Chennai");

visitedCities.add("Bangalore");

visitedCities.add("Hyderabad");

visitedCities.add("Chennai"); // Duplicate - will not be added again

visitedCities.add("Mumbai");

// Display the cities in visiting order

System.out.println("Recently visited cities:");

for (String city : visitedCities) {

System.out.println(city);

}

// Check if a city was visited

String searchCity = "Hyderabad";

if (visitedCities.contains(searchCity)) {

System.out.println(searchCity + " was visited.");

} else {

System.out.println(searchCity + " was not visited.");

}

}

}